Proyecto 1: Programación Dinámica y Voraz

William Franco Otero - 2259715
David Santiago Velasco Triana 2259479
Jhojan Stiven Castaño Jejen 22594776

Diciembre 2024

1 La Terminal Inteligente

1.1 Solución de Fuerza Bruta

En este apartado se presenta una solución de fuerza bruta para el problema de la Terminal Inteligente. El objetivo es transformar una cadena de caracteres en otra utilizando un conjunto de operaciones con distintos costos. A continuación, se detalla el algoritmo implementado en Python y su funcionamiento.

1.1.1 Descripción de la Metodología

La solución de fuerza bruta consiste en explorar todas las posibles secuencias de operaciones (avance, eliminación, reemplazo, inserción y eliminación total) para transformar una cadena x en otra y. El algoritmo recursivo explora todas las opciones y calcula los costos de cada operación, eligiendo la secuencia con el menor costo total. Las operaciones y sus respectivos costos son:

- **advance**: Mueve el cursor un carácter a la derecha. Costo: a
- **delete**: Borra el carácter bajo el cursor y mueve el cursor al siguiente carácter. Costo: d
- **replace**: Reemplaza el carácter bajo el cursor por otro y mueve el cursor una posición a la derecha. Costo: r
- **insert**: Inserta un nuevo carácter antes del carácter que está bajo el cursor. El cursor se mantiene en su posición. Costo: i
- **kill**: Borra los caracteres desde el que está bajo el cursor, incluyéndolo, hasta el final de la línea. Costo: k

El algoritmo utiliza una función recursiva que evalúa todas las secuencias posibles de operaciones y devuelve el costo total más bajo. La idea principal es simular todas las combinaciones de operaciones y luego seleccionar la que minimice el costo total.

1.1.2 Descripción del Código de Fuerza Bruta

El algoritmo recursivo que implementa la solución de fuerza bruta en Python realiza lo siguiente:

- La función `calcular_mejor_coste` es el núcleo del algoritmo, y es llamada recursivamente con diferentes estados de las cadenas `word1` y `word2`.
- En cada paso, la función evalúa si se debe avanzar (si los caracteres son iguales), eliminar un carácter, insertar uno nuevo, reemplazar un carácter o hacer un `kill` (eliminar hasta el final de la cadena).
- En cada uno de estos casos, el algoritmo calcula el costo de la operación y continúa llamando a la función recursivamente con el nuevo estado de las cadenas.
- La función también lleva un registro de las operaciones realizadas y los costos acumulados a través de las listas `operaciones` y `costos`.
- Al final, el algoritmo retorna la secuencia de operaciones y el costo total correspondiente a la secuencia más barata.

1.1.3 Ejemplo con Imágenes

Ejemplo del resultado de la bruta

Figure 1: Resultado de la fuerza bruta.

1.1.4 Complejidad Computacional

El código tiene una complejidad **exponencial** debido a la recursión que evalúa múltiples operaciones en cada subproblema. En cada llamada recursiva, el algoritmo evalúa 4 posibles operaciones, lo que da lugar a una expansión del número de subproblemas.

La complejidad es aproximadamente $O(4^{n \times m})$, donde n y m son las longitudes de las cadenas. Esto significa que el tiempo de ejecución crece rápidamente con el tamaño de las cadenas, lo que lo hace ineficiente para entradas grandes.

1.1.5 Tiempos y tiempo promedio

```
0.173631,0.169437,0.190481,0.177261,0.167970,0.182843,0.169887,
0.173186,0.181572,0.173189,0.161602,0.184029,0.169761,0.167580,
0.179259,0.172946,0.168865,0.180948,0.181617,0.163347,0.181278,
0.168318,0.162920,0.180605,0.171515,0.183517,0.180681,0.170275,
0.166386,0.182074,0.177125,0.167198,0.182190,0.170212,0.163987,
0.180182,0.170493,0.164240,0.180495,0.173340,0.166117,0.177882,
0.173614,0.164644,0.203581,0.175040,0.166485,0.182678,0.169147,
0.165166
```

Promedio de tiempo: 0.173285 segundos

1.1.6 Conclusión

La solución de fuerza bruta, aunque correcta, no es eficiente para cadenas largas debido a su complejidad exponencial. Este enfoque explora todas las posibles secuencias de operaciones, lo que garantiza encontrar la secuencia con el menor costo total, pero a un alto costo computacional.

1.2 Solución Voraz

En este apartado se presenta una solución **voraz** para el problema de la Terminal Inteligente. El objetivo es transformar una cadena de caracteres en otra utilizando un conjunto de operaciones con distintos costos. A continuación, se detalla el algoritmo implementado en Python y su funcionamiento.

1.2.1 Descripción de la Metodología

La solución **voraz** consiste en explorar las operaciones que minimizan el costo en cada paso, sin realizar una búsqueda exhaustiva. El algoritmo recursivo evalúa las posibles operaciones y calcula los costos de cada una, eligiendo la operación con el menor costo en cada paso y realizando la transformación de la cadena `source` en la cadena `objetivo`. Las operaciones y sus respectivos costos son:

- **advance**: Mueve el cursor un carácter a la derecha. Costo: `a`.
- **delete**: Borra el carácter bajo el cursor y mueve el cursor al siguiente carácter. Costo: `d`.
- **replace**: Reemplaza el carácter bajo el cursor por otro y mueve el cursor una posición a la derecha. Costo: `r`.
- **insert**: Inserta un nuevo carácter antes del carácter que está bajo el cursor. El cursor se mantiene en su posición. Costo: `i`.

- **kill**: Borra los caracteres desde el que está bajo el cursor, incluyéndolo, hasta el final de la línea. Costo: `k`.

El algoritmo utiliza una función recursiva que evalúa las posibles opciones de operaciones y devuelve el costo total más bajo. La idea principal es simular las combinaciones de operaciones y luego seleccionar la que minimice el costo total.

1.2.2 Descripción del Código de la Solución Voraz

El código implementa una solución basada en un enfoque **voraz**, donde se evalúa la mejor operación en cada paso. El algoritmo trabaja de la siguiente manera:

1. **Clase** `TransformationResult`: Esta clase almacena el resultado de la transformación, que incluye el costo total y las operaciones realizadas.
2. **Clase** `interfaz_voraz`:
   - **Atributos**:
     - `source`: La cadena original.
     - `objetivo`: La cadena de destino.
     - `costs`: Un diccionario con los costos asociados a cada operación.
   - **Método** `transform`:
     - Este método aplica las operaciones de transformación sobre la cadena `source` y calcula el costo total.
     - Utiliza un bucle que recorre ambas cadenas y realiza las transformaciones paso a paso, eligiendo la operación de menor costo.
   - **Método** `mejor_opcion`:
     - Este método evalúa las posibles operaciones (reemplazo, eliminación, inserción) y selecciona la que tiene el menor costo basado en las cadenas `source` y `objetivo`.
   - **Método** `letras_faltantes`:
     - Este método maneja los casos donde se alcanzan las posiciones finales de las cadenas. Si hay caracteres restantes en la cadena `objetivo`, se insertan. Si quedan caracteres en `source` pero no en `objetivo`, se utiliza la operación `kill`.

1.2.3 Ejemplo con Imágenes

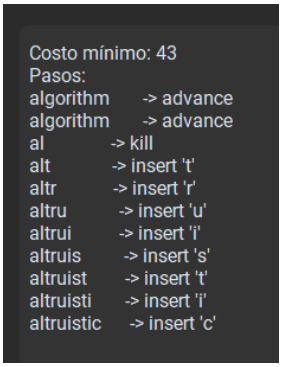

Figure 2: Resultado de la Voraz.

1.2.4 Complejidad Computacional

El algoritmo implementado no tiene una complejidad exponencial, como en el caso de la fuerza bruta, sino que es más eficiente porque utiliza un enfoque **voraz**. Aunque no explora todas las posibles secuencias de operaciones, el algoritmo realiza una única evaluación en cada paso y selecciona la operación con el menor costo en ese momento.

La complejidad de este enfoque es $\mathbf{O(n + m)}$, donde `n` es la longitud de `source` y `m` es la longitud de `objetivo`. En cada paso, el algoritmo realiza una comparación constante entre las operaciones posibles, lo que hace que el tiempo de ejecución dependa linealmente del tamaño de las cadenas.

1.2.5 Tiempos y tiempo promedio

```
0.031605,0.033091,0.032114,0.032111,0.030616,0.031272,0.032107,
0.032323,0.032535,0.032615,0.031599,0.032653,0.032516,0.032601,
0.031611,0.032021,0.032627,0.032122,0.033532,0.030664,0.032662,
0.032074,0.032019,0.032340,0.030598,0.031611,0.030787,0.032519,
0.031617,0.031541,0.033154,0.031616,0.032519,0.032164,0.032235,
0.032807,0.032622,0.031602,0.032680,0.033531,0.033662,0.032328,
0.032211,0.033023,0.033043,0.032593,0.032691,0.032091,0.032479,
0.033121
```

Promedio de tiempo: 0.032246 segundos.

1.2.6 Conclusión

La solución **voraz**, aunque no explora todas las posibles combinaciones de operaciones, es una opción eficiente para transformar cadenas cuando se busca una solución rápida. En casos con cadenas largas, este enfoque es mucho más adecuado que la fuerza bruta, ya que no requiere evaluar todas las secuencias posibles de operaciones. Sin embargo, no garantiza encontrar la solución óptima en todos los casos, ya que se basa en tomar decisiones locales (la mejor operación en cada paso) en lugar de explorar todas las opciones posibles.

1.3 Solución dinamica

En este apartado se presenta una solución más eficiente basada en Programación Dinámica para el problema de la Terminal Inteligente. Este enfoque reduce significativamente la complejidad computacional al evitar el cálculo redundante de subproblemas.

1.3.1 Descripción de la metodologia

La solución utiliza una tabla para almacenar los costos mínimos de transformar una cadena inicial en una cadena destino. Este enfoque evita la recalculación de subproblemas y asegura que cada operación sea evaluada solo una vez. Las operaciones consideradas son las siguientes:

- **Avance:** Mueve el cursor un carácter a la derecha.
- **Eliminación** Borra el carácter bajo el cursor.
- **Inserción:** Inserta un nuevo carácter antes del cursor.
- **Reemplazo:** Sustituye el carácter bajo el cursor por otro.
- **Reemplazo total:** Cambia la cadena completa a la cadena destino.

La tabla se construye llenando iterativamente los costos asociados a cada operación, comenzando por casos base.

1.3.2 Ejemplo con imagen

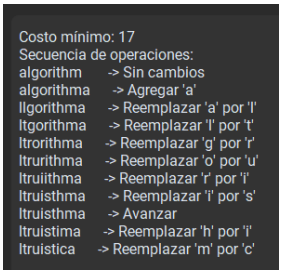

Figure 3: Resultado de la Dinamica.

1.3.3 Complejidad computacional

El enfoque de programación dinámica mejora drásticamente la eficiencia con respecto al enfoque de fuerza bruta. La complejidad del algoritmo es aproximadamente O(n×m), donde n y m son las longitudes de las cadenas inicial y destino, respectivamente. Este tiempo es mucho más manejable que la complejidad exponencial del método anterior.

1.3.4 Tiempos y tiempo promedio

```
0.032163, 0.031784, 0.033005, 0.032622, 0.032639, 0.032992, 0.032584,
0.032100, 0.032527, 0.031174, 0.037740, 0.031719, 0.031166, 0.032037,
0.032748, 0.033158, 0.032920, 0.033118, 0.032136, 0.032123, 0.032953,
0.031632, 0.033109, 0.032196, 0.033041, 0.031902, 0.031521, 0.032634,
0.033400, 0.048140, 0.032295, 0.032422, 0.034753, 0.033196, 0.031597,
0.032322, 0.031760, 0.031547, 0.032610, 0.032265, 0.031719, 0.032594,
0.032935, 0.032230, 0.031794, 0.032083, 0.031702, 0.032718, 0.032166, 0.032234
```

Tiempo promedio: 0.0328 segundos.

1.3.5 Conclusión

La solución basada en programación dinámica es significativamente más eficiente que el enfoque de fuerza bruta y asegura mas la opmitiamalidad que la voraz. Permite manejar cadenas de mayor longitud en tiempos razonables al aprovechar la reutilización de resultados intermedios. Sin embargo, aún existe una limitación en el uso de memoria debido a la construcción de la tabla.

1.4 Calcular el valor de una solución óptima

El proceso para calcular la solución óptima para transformar una cadena en otra a través de operaciones básicas (insertar, eliminar, reemplazar) se puede visualizar mediante el llenado de una matriz de programación dinámica. A continuación, se describe este proceso de manera más detallada, sin entrar en nombres específicos de variables de código, sino utilizando una explicación más general.

1.4.1 Construcción de la matriz de programación dinámica (dp)

Se comienza creando una matriz de tamaño $(n + 1) \times (m + 1)$, donde n es la longitud de la cadena de origen y m la longitud de la cadena de destino. La idea de esta matriz es almacenar los costos mínimos para transformar diferentes prefijos de la cadena de origen en diferentes prefijos de la cadena de destino.

La matriz se inicializa con valores muy grandes (simulando el infinito), a excepción de la celda inicial que corresponde a transformar dos cadenas vacías. Este valor es 0 porque no hay costo para transformar cadenas vacías.

1.4.2 Llenado de la matriz

1. **Primera fila y primera columna**:
   - La primera fila de la matriz se llena con los costos de insertar caracteres en la cadena vacía de la cadena de origen. Cada celda de esta fila representa el costo de insertar un número específico de caracteres.
   - La primera columna se llena con los costos de eliminar caracteres de la cadena de destino. Cada celda de esta columna refleja el costo de eliminar un número específico de caracteres.
2. **Relleno de las celdas restantes**:
   - Para cada celda que corresponde a una posición x en la cadena de origen y y en la cadena de destino, se evalúan las cuatro operaciones posibles (avanzar, reemplazar, insertar, eliminar).
     - **Avanzar**: Si los caracteres en la posición $x - 1$ de la cadena de origen y $y - 1$ de la cadena de destino son iguales, no se realiza ninguna operación y se toma el costo de avanzar.
     - **Reemplazar**: Si los caracteres son diferentes, se evalúa el costo de reemplazar el carácter de la cadena de origen por el de la cadena de destino.
     - **Insertar**: Se evalúa el costo de insertar un carácter de la cadena de destino en la cadena de origen.
     - **Eliminar**: Se evalúa el costo de eliminar un carácter de la cadena de origen.
   - En cada celda, se selecciona la operación con el costo mínimo de las opciones anteriores.

3. **Decisión de operaciones**:

   - Si los caracteres en las posiciones comparadas coinciden, el costo asociado es el mismo que el de la celda anterior, lo que refleja que no es necesario hacer ninguna operación.
   - Si no coinciden, se realiza un cálculo basado en la opción de reemplazar, que puede ser más barata que realizar una inserción o eliminación, dependiendo de los costos asociados.

1.4.3 Almacenamiento de las decisiones

A medida que se llena la matriz, también se registran las decisiones tomadas en cada celda. Estas decisiones se guardan en una matriz auxiliar que permite reconstruir la secuencia de operaciones realizadas, lo cual es útil para imprimir los pasos específicos que transforman una cadena en otra.

1.4.4 Resultado final

Al final del proceso, el valor óptimo de la transformación estará en la celda correspondiente a la longitud completa de ambas cadenas. Este valor contiene el costo mínimo necesario para transformar la cadena de origen en la cadena de destino.

Además de calcular el costo, la matriz de operaciones también nos proporciona las acciones específicas que se tomaron en cada paso para obtener esta solución óptima. Esto permite reconstruir las operaciones exactas realizadas (inserciones, eliminaciones, reemplazos) en el orden correcto.

2 El Problema De La Subasta Publica

2.1 Solución Fuerza Bruta

2.1.1 Descripción de la Metodología

La solución de fuerza bruta explora todas las combinaciones posibles de asignación de acciones entre los oferentes y el gobierno, evaluando cada combinación para encontrar la que maximice el ingreso total. Este enfoque garantiza encontrar la solución óptima, pero a un alto costo computacional debido al crecimiento exponencial de las combinaciones.

2.1.2 Descripción del Código

El algoritmo recorre todas las combinaciones posibles mediante recursión:

- Genera combinaciones de asignación respetando las restricciones de cada oferente (`mi` y `Mi`).

- Calcula el ingreso total para cada combinación, considerando tanto las acciones asignadas a los oferentes como las que sobran, asignadas al gobierno.
- Almacena la combinación con el mayor ingreso total y la retorna como resultado final.

2.1.3 Ejemplo con Imágenes

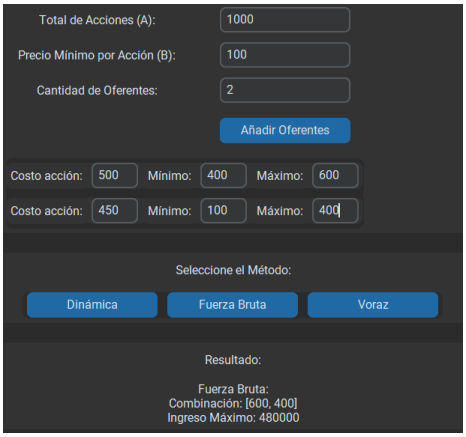

Figure 4: Resultado de la Bruta con solo 2 oferentes(mas se demora demasiado).

2.1.4 Complejidad Computacional

La complejidad de este enfoque es $O(n \cdot M^n)$, donde n es el número de oferentes y M es el rango promedio de asignaciones posibles por oferente $(Mi - mi)$. Esto lo hace ineficiente para casos con muchos oferentes o amplios rangos de asignación.

2.1.5 Tiempos y Tiempo Promedio

3.987648, 4.015759, 3.888202, 3.821068, 3.856492, 3.873959, 3.900323, 3.891591, 3.875697, 3.816374, 3.844572, 3.907035, 3.886005, 3.917212, 3.941003, 3.884036, 3.917240, 3.911512, 3.908054, 3.894738, 3.909533, 3.912472, 3.917040, 3.942291, 3.898398, 3.908240, 3.937117, 3.927024, 3.930264, 3.917647, 3.947821, 3.962085, 3.921453, 3.951044, 3.944247, 3.901383, 3.965267, 3.902938, 3.932192, 3.915539, 3.887567, 3.923954, 3.918888, 3.861634, 3.889941, 3.860557, 3.858771, 3.882470, 3.853969, 3.908837

Tiempo promedio: 3.906582 segundos.

2.1.6 Conclusión

La solución por fuerza bruta es óptima, ya que evalúa todas las combinaciones posibles, pero su complejidad la hace poco práctica para problemas de mayor tamaño. Es útil principalmente como referencia para comparar con métodos más eficientes.

2.2 Solución Voraz

2.2.1 Descripción de la Metodología

La solución voraz asigna acciones de manera iterativa, seleccionando en cada paso al oferente con el mayor precio por acción (p_i), respetando sus restricciones (mi, Mi). Una vez asignadas las acciones a los oferentes, las restantes se asignan al gobierno.

2.2.2 Descripción del Código

El algoritmo voraz sigue estos pasos:

- Ordena a los oferentes de forma descendente según su precio por acción (p_i).
- Asigna acciones al oferente más valioso posible hasta cumplir con las restricciones o agotar las acciones disponibles.
- Continúa con el siguiente oferente en el orden hasta distribuir todas las acciones.

2.2.3 Ejemplo con Imágenes

Figure 5: Resultado de la voraz.

2.2.4 Complejidad Computacional

El algoritmo tiene una complejidad de $O(n \log n)$ debido a la ordenación inicial de los oferentes. La asignación de acciones tiene una complejidad lineal respecto al número de oferentes, $O(n)$. En total, la complejidad es $O(n \log n)$.

2.2.5 Tiempos y Tiempo Promedio

0.03269, 0.031201, 0.033086, 0.032961, 0.032524, 0.031853, 0.03706, 0.032568, 0.03208, 0.032026, 0.031546, 0.031754, 0.032252, 0.031162, 0.031664, 0.030881, 0.031971, 0.031706, 0.032109, 0.032583, 0.031802, 0.031583, 0.035069, 0.032112, 0.032577, 0.031519, 0.032578, 0.033028, 0.032187, 0.032237, 0.032727, 0.031565, 0.032528, 0.032595, 0.031057, 0.032571, 0.031245, 0.032608, 0.032057, 0.032567, 0.031615, 0.032527, 0.032562, 0.031541, 0.031597, 0.031599, 0.03208, 0.032574, 0.031083, 0.032049

Tiempo promedio: 0.032222 segundos.

2.2.6 Conclusión

La solución voraz es eficiente y adecuada para problemas donde las restricciones son simples y el criterio de selección es claramente determinante. Sin embargo, no garantiza encontrar la solución óptima en todos los casos.

2.3 Solución Dinámica

2.3.1 Descripción de la Metodología

La solución dinámica utiliza una tabla para almacenar el ingreso máximo alcanzable con cada subconjunto de oferentes y diferentes cantidades de acciones disponibles. Este enfoque asegura la optimalidad aprovechando los subproblemas solapados y evita la evaluación redundante.

2.3.2 Descripción del Código

El algoritmo sigue estos pasos:

- Crea una matriz $dp[i][j]$, donde i representa los oferentes considerados y j la cantidad de acciones disponibles.
- Llena la tabla iterativamente:
  - Para cada oferente y cada posible cantidad de acciones, calcula el ingreso máximo al incluir o excluir al oferente.
  - Se consideran todas las posibles asignaciones del oferente dentro de sus límites (mi, Mi).

- Reconstruye la solución óptima a partir de la tabla, rastreando las decisiones tomadas.

2.3.3 Ejemplo con Imágenes

Figure 6: Resultado de la dinam.

2.3.4 Complejidad Computacional

La complejidad del algoritmo es $O(n \cdot A \cdot M)$, donde n es el número de oferentes, A es el total de acciones disponibles, y M es el rango promedio de asignaciones por oferente $(Mi - mi)$.

2.3.5 Tiempos y Tiempo Promedio

0.157187, 0.15673, 0.156635, 0.157056, 0.15691, 0.154895, 0.156105, 0.159172, 0.162761, 0.155791, 0.153718, 0.158648, 0.158006, 0.156606, 0.157253, 0.154088, 0.155885, 0.156188, 0.155904, 0.154941, 0.164936, 0.155124, 0.156101, 0.161533, 0.15714, 0.156749, 0.153582, 0.154837, 0.159127, 0.156496, 0.157123, 0.154565, 0.156177, 0.156307, 0.154579, 0.157074, 0.165726, 0.156909, 0.155194, 0.15594, 0.161119, 0.158794, 0.162677, 0.156849, 0.156332, 0.15633, 0.158727, 0.155915, 0.157057

Tiempo promedio: 0.157214 segundos.

2.3.6 Conclusión

La solución dinámica garantiza la optimalidad con una complejidad mucho más manejable que la fuerza bruta. Sin embargo, el uso de memoria puede ser una limitación para problemas de gran escala.

2.4 Conclusión General

Comparando los tres métodos:

- **Fuerza Bruta:** Garantiza la solución óptima, pero tiene una complejidad exponencial y es ineficiente para problemas grandes.
- **Voraz:** Es rápido y eficiente ($O(n \log n)$), pero no siempre encuentra la solución óptima.
- **Dinámica:** Asegura la optimalidad con una complejidad más razonable ($O(n \cdot A \cdot M)$), aunque requiere mayor uso de memoria.

En general, la programación dinámica es la mejor opción para este problema cuando la optimalidad es crítica y los recursos de memoria lo permiten. La solución voraz es una alternativa práctica para casos más simples.